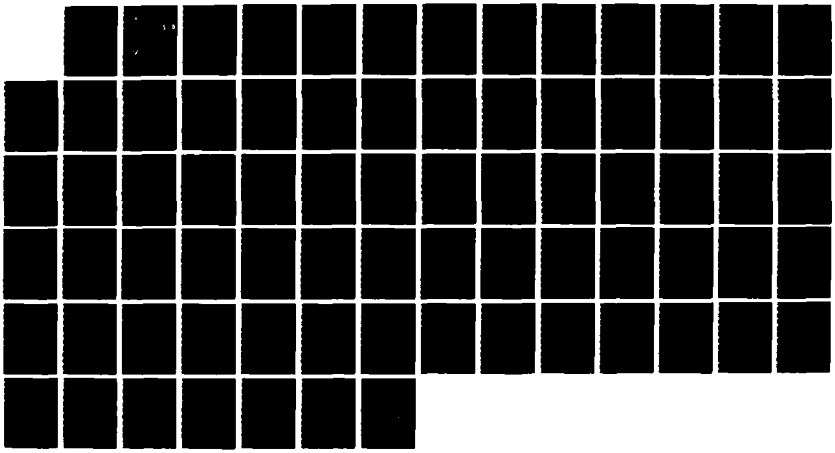
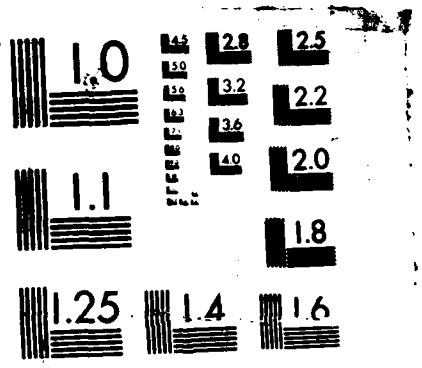
AD-A180 021 ADDASET (AIR DEFENSE DIGITAL AVIONICS SEEKER ENHANCEMENT TECHNOLOGY) EVAL. (U) ALABAMA UNIV IN HUNTSVILLE SCHOOL OF ENGINEERING P L ROMINE ET AL 1/1

UNCLASSIFIED 05 JAN 87 UAH-5-32329 AMSMI-CR-RD-SS-86-4 F/G 16/4 2 NL





MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

2

TECHNICAL REPORT CR-RD-SS-86-4

ADDASET EVALUATION/CALIBRATION SUPPORT

Prepared By:

Peter L. Romine
Timothy A. Palmer
Terry N. Long
School of Engineering
The University of Alabama in Huntsville
Huntsville, AL 35899

Prepared For:

Systems Simulation and Development Directorate
Research, Development, and Engineering Center
US Army MICOM

JANUARY 1987

U.S. ARMY MISSILE COMMAND
Redstone Arsenal, Alabama 35898-5000

Cleared for public release; distribution is unlimited.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A180 021

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188
Exp. Date Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Cleared for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): 5-32329	5. MONITORING ORGANIZATION REPORT NUMBER(S): CR-RD-SS-86-4
6a. NAME OF PERFORMING ORGANIZATION: UAH School of Engineering; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: Commander, US Army MICOM ATTN: AMSMI-RD-SS-SE
6c. ADDRESS (City, State, and ZIP Code): The University of Alabama in Huntsville Huntsville, AL 35899	7b. ADDRESS (City, State, and ZIP Code): Redstone Arsenal, AL 35898
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: DAAH01-82-D-A008
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)

ADDASET Evaluation/Calibration Support

12. PERSONAL AUTHOR(S)

Peter L. Romine, Timothy A. Palmer, and Terry N. Long

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 11/84 TO 5/85	87/Jan/05	90

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

ADDASET, DEBUG, Digital Avionics, Program Listings, Seeker, User Aids

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Verification and calibration software for the Air Defense Digital Avionics Seeker Enhancement Technology (ADDASET) hardware is documented. Software rationale, operation sequence, program listings and user aids are provided.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. M. M. Hallum	22b TELEPHONE (Include Area Code): 205-876-4141; 22c OFFICE SYMBOL: AMSMI-RD-SS

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

CR-RD-SS-86-4 January 1987

ADDASET EVALUATION/CALIBRATION SUPPORT

Final Technical Report for Period
28 November 1984 through 31 May 1985

January 1987

Prepared By:

Peter L. Romine
Timothy A. Palmer
Terry N. Long
School of Engineering
The University of Alabama in Huntsville
Huntsville, AL 35899

Prepared For:

Systems Simulation and Development Directorate
Research, Development, and Engineering Center
US Army MICOM

TABLE OF CONTENTS

PREFACE

This technical report was prepared by the Research Staff of the Electrical and Computer Engineering Department, School of Engineering, The University of Alabama in Huntsville. The work documented in this report was performed by Terry N. Long, Timothy A. Palmer, and Peter L. Romine. The purpose of this report is to provide documentation of technical work performed and results obtained under delivery order number 0044, contract number DAAH01-82-D-A008. Mr. Terry N. Long was the principal investigator; Dr. M. M. Hallum, III, Chief, Systems Evaluation Branch, was the technical monitor; and Mr. Mark Horton also from the Systems Evaluation Branch of the Systems Simulation and Development Directorate, U.S. Army Missile Command, provided technical coordination.

The technical viewpoints, opinions, and conclusions expressed in this document are those of the authors and do not necessarily express or imply policies or positions of the U.S. Army Missile Command.

I. INTRODUCTION

A comprehensive test routine was required for verification and calibration of the Air Defense Digital Avionics Seeker Enhancement Technology (ADDASET) hardware built by the Boeing Company. A C-routine, DEBUG, and its supporting routines provide calibration and debug capabilities for the HAWK/ADDASET test bed. The programs were generated using a Zilog System-8000 UNIX-based development system which was chosen for compatibility with the Z8002-based single board computers used in the ADDASET system. Compatibility is such that compiled object modules and executables originating on the System-8000 can be transported directly to the computers in the ADDASET system.

Operation of the DEBUG test program, which is a very versatile multipurpose program, is described in Section II. Additional special purpose routines were developed during the hardware acceptance phase of the ADDASET program and are briefly described in Section III. Several user aids were also developed or identified during the acceptance process, and are described in Section IV. Conclusions and recommendations are presented in Section V.

II. DEBUG TEST PROGRAM

A. Introduction

The DEBUG test program is a versatile, multipurpose diagnostic program that can be used for operational verification or calibration of ADDASET hardware. Program execution is initiated after entering "debug". The program is fully menu-driven with the menu illustrated in Table 1 being displayed initially. As shown by the menu, five separate processes can be performed. The desired process can be chosen by the proper selection. Output and input discretes can be controlled as well as analog-to-digital converters (A/D's) and digital-to-analog converters (D/A's). There is also a routine to facilitate calibration of the system's analog buffers.

Discrete I/O control, analog I/O control, and analog buffer card calibration are described below. Additionally, some notes concerning DEBUG software design are provided.

TABLE 1. Opening Menu

```
O)  Output byte to discrete channel
I)  Input byte from discrete channel
A)  A/D input routine
D)  D/A output routine
C)  Calibrate analog-buffer card
E)  Exit to monitor

Enter selection (O,I,A,D,C,E):
```

B. Discrete I/O Control

The ADDASET system currently supports three 8-bit discrete input and three 8-bit discrete output channels. The digital interface is required for monitoring and controlling the four binary signals originating on the missile (Radar Enable, End of Sustain, Pitch Precession, and Antenna Center commands) and the eight software emulated signals (Radar Enable, Head Enable, Elevon Enable, Test Mode, Test Select 1, Test Select 2, Analog Stabilization Loop Closure, and Antenna Center Mode). Each discrete has a corresponding LED indicator on the front panel of the interface rack to display the status of each bit. By selecting the Input/Output byte option from the main menu, the operator can either read discrete input channel A or specify an arbitrary bit pattern for output on a selected channel.

Press "O" or "o" to output a byte on a discrete channel. The system will prompt the user for the channel number (0-2) and the data in hex (0-FF). After entry, the output data is also echoed to the terminal screen.

Press "I" or "i" to input data from discrete channel A. Input data is displayed on the terminal screen and updated continuously. The input operation can be halted at any time by pressing the ESC key on the terminal keyboard.

C. Analog I/O Control

The A/D and D/A options allow the operator to monitor the voltage at any of 24 analog inputs or specify an arbitrary voltage for output on any of 16 analog outputs. Voltages are displayed on the terminal screen in floating point. Real valued voltages can also be entered at the keyboard for output on a selected analog line.

Press "A" or "a" to convert an external analog voltage signal to digital for display on the terminal screen. The operator is prompted to select an analog input channel (0-23). The voltage is then displayed on the terminal screen and updated continuously until the ESC key is pressed.

Press "D" or "d" to output an arbitrary voltage. The operator is prompted to enter the channel number (0-15) and the output data. Output data can be any real number voltage within the limits of the power supply ($\pm$10V). The actual output value is echoed to the terminal after output on the selected channel.

D. Calibrate Analog-Buffer Card

The analog-buffer card is the analog interface between the missile and the ADDASET rack. The buffer card's primary function is for adjustment of biases and gains associated with each analog I/O line on the interface rack. It also provides protection for missile and rack components. To aid in calibration of the analog-buffer card, the calibration option provides the menu shown in Table 2.

TABLE 2. Analog-Buffer Card Calibration Menu

G)	select ground
+)	select plus(+) voltage
-)	select minus(-) voltage
D)	display A/D channels
M)	main menu

Select ground, plus voltage, and minus voltage options apply the corresponding signal to each of the analog inputs. The converted voltage can then be monitored using the display A/D function. In this configuration, the analog card can be calibrated by adjusting the gains and/or biases on the card until the correct voltage is displayed on the terminal. By pressing "G", "+", or "-", the inputs to the buffer card are either grounded, set to the precision positive calibration voltage, or set to the precision negative calibration voltage.

To monitor a particular channel, press "D" or "d" and input the desired A/D channel number (0-15) when prompted. The voltage presented to the selected channel is displayed on the terminal screen and is updated continuously until the ESC key is pressed. The escape key returns the user to the configuration menu.

Press "M" or "m" to exit the calibration menu and return to the opening menu.

E. DEBUG Software Notes

A listing of DEBUG code is presented in Appendix A. Some important pieces of information can be derived from this code. First, the program's menu format can be examined by examining the arguments of the "mess-c" functions. Also, the register addresses for I/O and control functions can be easily found in the code.

A large portion of the code was developed to facilitate manipulation of floating point numbers. Some of the routines are directly callable from C and some are not. It is useful to note that they can be used for a variety of applications.

III. ADDITIONAL I/O TEST PROGRAMS

DEBUG resulted from the combination of a number of test and calibration programs. Several other routines were developed in addition to those included in DEBUG and they perform a variety of useful functions.

1. ADDISP

ADDISP is particularly useful because it provides for the simultaneous display of up to seven of the 23 A/D channels. The user must supply the number of channels to be displayed at one time (1-7, do not choose 0). The user must also supply which ADC (1-23) should be displayed in each column (0-6) of the display. The user can then choose continuous updates (press C), discrete updates (press D), or pause (press P). A listing of ADDISP is provided in Appendix B.

2. DARAMP

DARAMP is also very useful as a verification tool. It generates a ramp output to a block of D/A's. The user must supply the start channel and stop channel of the block of D/A's to be exercised as well as the start value and stop value of the ramp. The user must also supply the step size of the ramp input divided by sixteen. Dynamic operation of the D/A's can be examined by using this routine, and each discrete level can be evaluated. A listing of DARAMP is provided in Appendix C.

3. ADTODA

ADTODA facilitates simultaneous output to a D/A channel from an A/D channel. This permits operation verification by tying pairs of channels together. A listing of ADTODA is provided in Appendix D.

4. DSOUT

DSOUT simply sequences all of the output discretes. The program is marginally useful, and its listing is provided in Appendix E.

5. ACTIVITY

ACTIVITY exercises the operation modes. It is marginally useful, and a listing of the program is provided in Appendix F.

IV. EVALUATION/CALIBRATION AIDS

A detailed review of ADDASET documentation revealed that wiring information exists for all system interfaces. However, the information exists in various forms with no way to easily trace a signal from beginning to end. Figure 1 was developed in an effort to summarize the configuration and to provide a starting point for the wiring documentation. ADDASET/HAWK system elements are identified along with individual connectors. Each connector has been assigned a number, and they are provided in the figure. Additionally, the types of wiring and the numbers of lines have been identified. The most useful characteristic of the figure is the identification of the documentation that corresponds to each interface. The amount of interface documented by each piece of documentation and the set numbers of the corresponding documentation are also noted.

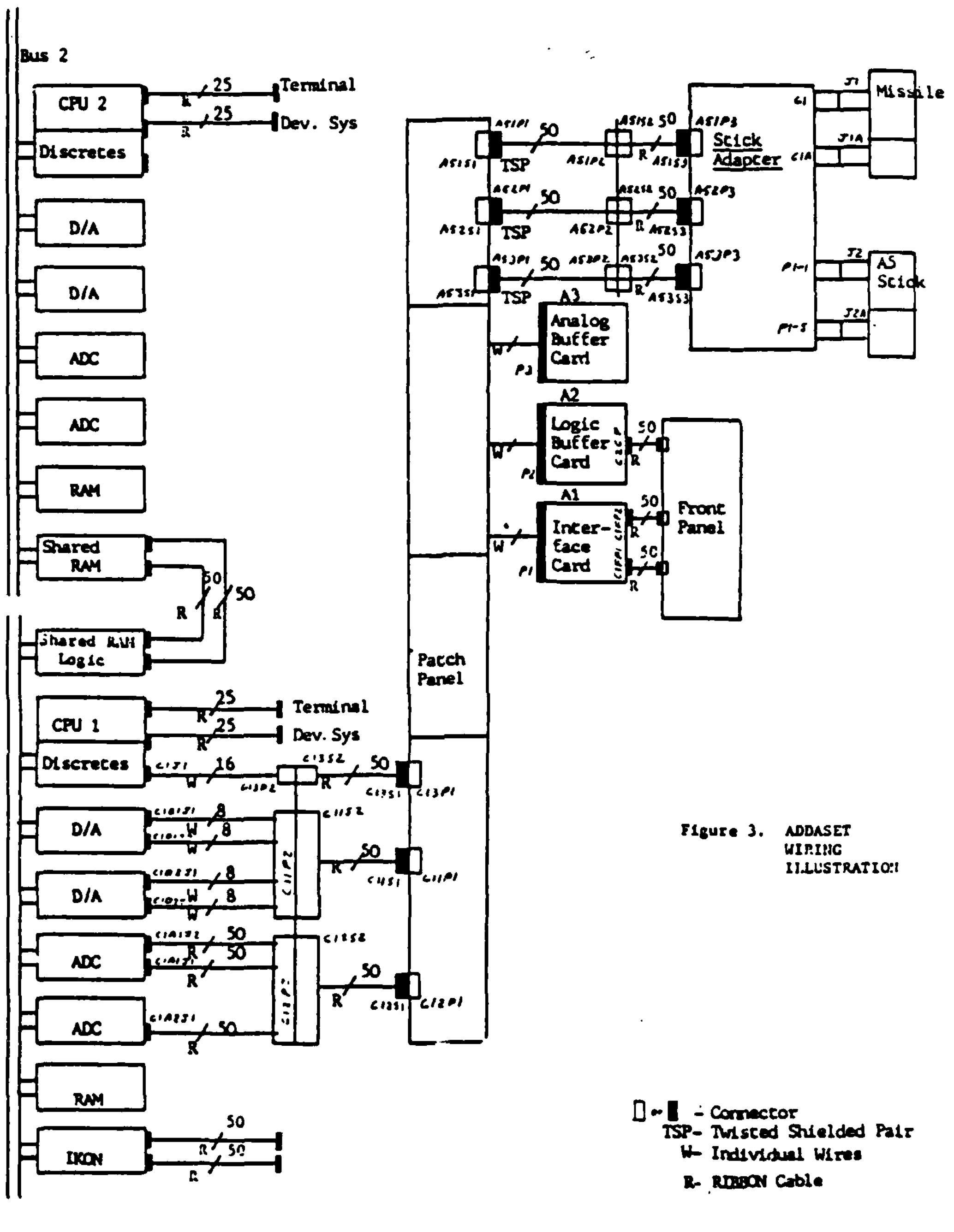


Figure 1. ADDASET wiring illustration.

Table 3. A/D and D/A Scaling

Hex	Volts	Decimal
7FF0	+10	2047
7320	+9	1842
6660	+8	1638
5990	+7	1433
4CC0	+6	1228
3FF0	+5	1023
3330	+4	0819
2660	+3	0614
1990	+2	0409
0CD0	+1	0205
0000	0	0
F330	-1	-0205
E670	-2	-0409
D9A0	-3	-0614
CCD0	-4	-0819
C010	-5	-1023
B340	-6	-1228
A670	-7	-1433
99A0	-8	-1638
8CE0	-9	-1842
8010	-10	-2047

Another verification aid is provided in Table 3. It contains scaling numbers used by the A/D's and D/A's. However, its usefulness may be marginal since scaling is now provided within DEBUG code.

V. CONCLUSIONS AND RECOMMENDATIONS

The programs and aids described in this report provide a comprehensive means to verify ADDASET hardware operation and provide ADDASET calibration. They have been used in the acceptance testing phase of the ADDASET program and have been used to calibrate the analog components of the system. They have also been used to monitor missile discretes in the various modes of operation.

The C language used for program development has proved to be very advantageous for I/O type activities. Therefore, it is recommended that the language be used for all ADDASET programming which requires user or hardware interfaces.

The System-8000 development system used for program development has presented several difficulties. Consequently, it is recommended for alternative systems to be examined for program development. PC type systems should be carefully considered.

The ADDASET program now has the test bed required to begin the development of digital avionics techniques and the aids required to maintain that test bed. Therefore, the ADDASET program should proceed at a rapid pace.

APPENDIX A

DEBUG

```
/***********************************************************************

        Main program to allow user to modify D/A and output discretes a
        as monitor A/D and input discretes.

        Functions defined in debug.c module:

                    main()              C routine provides for the men
                                        user interface.


         Externals  { in_char,out_crlf,mess_c,in_int,out_int,da_out,cio
                      cio_outa, cio_outb, cio_outc, cio_ina,ad_in_a,ad_
                      fp_in,fp_out,fp_out_e,fpcon,fpconv,fdiv }



***********************************************************************

extern  int      in_char(),out_crlf(),in_int(),out_int(),da_out();
extern  int      cio_init(),cio_outa(),cio_outb(),cio_outc(),cio_ina();
extern  int      ad_in_a(),ad_in_b();
extern  int      mess_c();
extern  long int         fp_in(),fp_out(),fp_out_e(),fpcon(),fpconv(),
                         fdiv();

        int              i,j,k,l,chan,data;
        int              array[30],*point;
        char             c,d,e,go,ok;
        long int         con3276p8,long1,long2;

main () {
        con3276p8 = fpcon("3276.8");

        cio_init(1,4000); /* sets up cio for input to cio2a and output
                    /* cio1a,cio1b,cio1c */

        out_crlf(); /* clear a little bit of screen off */
        out_crlf();

        mess_c("General Test Program V1.0 ");

        go = 'Y';
        while ( go == 'Y') {
        out_crlf();
        out_crlf();
```

```
mess_c("O - output byte to discrete channel\n\r");
mess_c("I - input byte from discrete channel\n\r");
mess_c("A - A/D input routine\n\r");
mess_c("D - D/A output routine\n\r");
mess_c("C - Calibrate analog-buffer card\n\r");
mess_c("E - Exit to monitor ");

ok = 'N';
while ( ok != 'Y') {
        out_crlf();
        mess_c("Enter selection (O,I,A,D,C,E)  :");
        c = wait_char();
if ((c == 'O') | (c == 'I') | (c == 'A') | (c == 'D')) ok = 'Y
if ((c == 'C') | (c=='E')) ok='Y';
        }

out_crlf();
switch(c) {
        case 'O' : {
                out_crlf();
                mess_c("Enter discrete channel number (0-2) :"
                ok='N';
                while ( ok != 'Y') {
                        chan = in_dec();
                        if ((chan <= 2) & (chan >= 0)) ok ='Y'
                        if (ok == 'N') {
                                out_char(0x07);
                                mess_c("Invalid channel. ");
                                mess_c("Try again ");
                                out_crlf();
                                } /* if */
                        } /* while */
                out_crlf();
                ok='N';
                while (ok != 'Y') {
                        mess_c("Enter channel ");
                        out_int(chan);
                        mess_c(" data (0-FF) :");
                        data=in_int();
                        if ((data >= 0) & (data <= 256)) ok='Y
                        } /* while */
                out_crlf();
                switch (chan) {
                        case 0: cio_outa(data);
                        break;
                        case 1: cio_outb(data);
                                break;
                        case 2: cio_outc(data);
                                break;
                        default: break;
                        } /* switch */
                out_crlf();
                mess_c("Data (");
                out_int(data);
                mess_c(") present on channel ");
                out_int(chan);
                out_crlf();
                break;
                }


        case 'I': {
                out_crlf();
                out_crlf();
                mess_c("Press ESCape to exit ");
                out_crlf();
                out_crlf();
                mess_c("Input discrete data ");
                out_crlf();
                mess_c("b0 b1 b2 b3 b4 b5 b6 b7");
                out_crlf();
                while ((c=in_char()) != 0x1b) {
                        data = cio_ina();
                        out_char(0x0d);
                        j=1;
                        while ( j <= 128) {
                                k = (j & data);
                                if (k ==0) mess_c("0  ");
                                if (k !=0) mess_c("1  ");
                                j *= 2;
                                }
                        }
                out_crlf();
                break;
                }

        case 'A': {
                out_crlf();
                ok='N';
                while (ok != 'Y') {
                        mess_c("Enter channel number (0-23) :"
                        chan=in_dec();
                        if ((chan <= 23) & (chan >= 0)) ok='Y'
                        } /* while */

                out_crlf();
                mess_c("Press ESCape to exit ");
                out_crlf();
                out_crlf();
                mess_c("A/D channel ");
                out_int(chan);
                out_crlf();
```

```
          while ((c=in_char()) != 0x1b) {
                  point=array;
                  ad_in_a(point,1);
                  point=array+12;
                  ad_in_b(point,1);
                  data=array[chan];
                  out_char(0x0d);
                  out_int(data);
                  long2=data;
                  out_char(' ');
                  fp_out(fdiv(fpconv(long2),con3276p8));
                  }
          out_crlf();
          break;
          }

  case 'D' : {
          out_crlf();
          ok ='N';
          while (ok != 'Y') {
                  mess_c("Enter channel number (0-15) :"
                  chan=in_dec();
                  out_crlf();
                  if ((chan >= 0) & (chan <= 15)) ok ='Y
                  if (ok == 'N') {
                          out_char(0x07);
                          mess_c("Invalid channel number
                          mess_c("Try again");
                          out_crlf();
                          }
                  }
                  out_crlf();
                  out_crlf();
                  mess_c("Enter data for channel ");
                  out_int(chan);
                  mess_c(" :");
                  data=in_int();
                  array[0]=data;
                  point=array;
                  da_out(point,chan,1);
                  mess_c("Data (");
                  out_int(array[0]);
                  mess_c(") present on channel ");
                  out_int(chan);
                  out_crlf();
                  break;
                  }
```

```
case 'C' : {
        while(1) {
                mess_c("\n\n\r");
                mess_c("Analog buffer card calibration
                mess_c("\n\r");
                mess_c("G - select ground\n\r");
                mess_c("+ - select plus voltage\n\r");
                mess_c("- - select minus voltage\n\r")
                mess_c("D - display A/D channels\n\r")
                mess_c("M - main menu");
                c = wait_char();
                if (c=='M') break;
                switch (c) {
                case 'G': {
                        cio_outa(0x48);
                        mess_c("\n\rTest mode: Ground\
                        break;
                        }
                case '+': {
                        cio_outa(0x78);
                        mess_c("\n\rTest mode: Plus\n\
                        break;
                        }
                case '-': {
                        cio_outa(0x58);
                        mess_c("\n\rTest mode: Minus\n
                        break;
                        }
                case 'D': {
                        mess_c("\n\rChannel :");
                        i=in_dec();
                        if (i > 15) {
                           mess_c("\n\rToo big\n\r");
                           break;
                           }
                        if (i < 0) {
                           mess_c("\n\rToo small\n\r")
                           break;
                           }
                    mess_c("\n\rPress EScape to exit.\
                        while ((c=in_char()) != 0x1b)
                                point=array;
                                ad_in_a(point,1);
                                point=array+12;
                                ad_in_b(point,1);
                                data=array[i];
                                long2=data;
                                long2=fpconv(long2);
                                long2=fdiv(long2,con32
                                fp_out(long2);
                                out_char(0x0d);
                                } /* while */
                        break;
                        } /* case 'D' */
                        } /* switch */
                        } /* while */
                break;
                } /* case 'C' */

        default: go='N';
        } /* switch */
        } /* while */
} /* main */
```

Functions defined in inout_c.s module:

in_char	Assembly routine to check whether a character available in the receive buffer register of th UART from the keyboard. ENTRY: none EXIT: RL0 = 0, if no character was available RL0 = character, otherwise
wait_char	Assembly routine to wait for a character to be available from the keyboard. ENTRY: none EXIT: RL0 = character
out_char	Assembly routine to transmit one character to console port (terminal). ENTRY: RL7 = character to output EXIT: none
out_crlf	Assembly routine to transmit a <CR><LF> to the console port (terminal) ENTRY: none EXIT: none
mess_c	Assembly routine to transmit a sequence of nul terminated characters to the console port (ter ENTRY: R7 = address of start of string buffer EXIT: none
out_int	Assembly routine to convert a 16-bit integer t ascii-hex and send the characters to the conso port (terminal). ENTRY: R7 = 16-bit integer to print EXIT: none
in_int	Assembly routine to accept a 16-bit hex intege constant from the keyboard. The characters are echoed to the console port (terminal) as they input. Any out-of-band characters cause an err message to be displayed and the conversion is again. ENTRY: none EXIT: R2 = 16-bit result

Externals (NONE)

```
_inout_c MODULE

! RAM ALLOCATION CONSTANTS !

CONSTANT

! SCC REGISTER ADDRESSES !
SCCOA    :=      %FC21
SCCDA    :=      %FC31

GLOBAL

_in_char PROCEDURE
ENTRY

! INPUT A CHARACTER FROM THE TERMINAL !

TINPUT: INB     RL0,SCCOA                        !CHAR RECEIVED?!
        BITB    RL0,#0
        JR      Z,TDONE                          !JP IF NOT!
        INB     RL0,SCCDA                        !INPUT CHAR!
        RESB    RL0,#7                           !CLEAR PARITY BIT!
        LDB     RL2,RL0
        CLRB    RH2
        RET
TDONE:  CLR     R2
        RET
END _in_char


_wait_char PROCEDURE
ENTRY

! INPUT A CHARACTER FROM THE TERMINAL !

TWAIT:  INB     RL0,SCCOA                        !CHAR RECEIVED?!
        BITB    RL0,#0
        JR      Z,TWAIT                          !JP IF NOT!
        INB     RL0,SCCDA                        !INPUT CHAR!
        RESB    RL0,#7                           !CLEAR PARITY BIT!
        LDB     RL2,RL0
        CLRB    RH2
        RET
END _wait_char
```

```
_out_char PROCEDURE
ENTRY
        LDB     RL0,RL7

! OUTPUT A CHARACTER TO THE TERMINAL !

TOUTCH: PUSH    @R15,R1
        LDAR    R1,$+6
        JR      TOCHNS
        POP     R1,@R15
        RET

! OUTPUT CHARACTER TO TERMINAL WITHOUT USING RAM !

TOCHNS: INB     RH0,SCC0A
        BITB    RH0,#2
        JR      Z,TOCHNS
TOUT10: OUTB    SCCDA,RL0
        JP      @R1
END _out_char


_out_crlf PROCEDURE
ENTRY
        LDB     RL0,#'%R'
        CALR    TOUTCH
        LDB     RL0,#'%L'
        JR      TOUTCH
END _out_crlf


_mess_c PROCEDURE

ENTRY

print2: ld      r6,@r7
        inc     r7,#2
        ldb     rl1,rh6
        cpb     rl1,#0
        ret     z
        calr    print3
        ldb     rl1,rl6
        cpb     rl1,#0
        ret     z
        calr    print3
        jp      print2
print3: inb     rh0,SCC0A
        bitb    rh0,#2
        jr      z,print3
        outb    SCCDA,rl1
        ret
END _mess_c
```

```
_out_int PROCEDURE
ENTRY
        ldb     rl1,rh7
        call    out_byte
        ldb     rl1,rl7
        call    out_byte
        ret
out_byte:
        ldb     rl0,rl1
        andb    rl0,#%0f0
        srlb    rl0,#4
        call    out_nib
        ldb     rl0,rl1
        andb    rl0,#%0f
        call    out_nib
        ret
out_nib:
        addb    rl0,#%0030
        cpb     rl0,#%3a
        jr      c,out_nib2
        addb    rl0,#%07
out_nib2:
        inb     rh0,SCCOA
        bitb    rh0,#2
        jr      z,out_nib2
        outb    SCCDA,rl0
        ret
END _out_int
```

```
_in_int PROCEDURE
ENTRY

! this routine accepts a 16 bit integer (unsigned) from the console !
! and converts it to binary !

ii0:    clr     r8                      ! clear the result registers !
        clr     r9
ii1:    call    _wait_char
        ld      r7,r2                   ! echo character !
        call    _out_char               ! to console !
        cpb     rl2,#%0d                ! was it a CR ? !
        jr      nz,ii2                  ! continue if not !
        ld      r2,r8                   ! else, put result in r2 !
        ret                             ! and return !
ii2:    call    atob                    ! convert ascii to binary digi
        jr      c,ii3                   ! go if error !
        sll     r8,#4                   ! shift left 4 places !
        add     r8,r2                   ! add current digit to accumul
        jp      ii1                     ! go for more !
atob:   ldb     rh2,#0                  ! zero top part of rl2 !
        cpb     rl2,#'0'                ! make sure not too small !
        jr      c,atob2                 ! go if too small !
        cpb     rl2,#'9'+1              ! see if alpha offset required
        jr      c,atob3                 ! go if not required !
        cpb     rl2,#'A'                ! see if between 9 and A !
        jr      c,atob2                 ! go if so !
        cpb     rl2,#'F'+1              ! see if > F !
        jr      nc,atob2                ! go if so !
        subb    rl2,#%37                ! else, subtract alpha offset
        resflg  c                       ! reset error flag !
        ret                             ! and return !
atob2:  setflg  c                       ! indicate error !
        ret                             ! and return !
atob3:  subb    rl2,#%30
        resflg  c
        ret                             ! and return !
ii3:    lda     r7,II4                  ! tell user to try again !
        call    _mess_c                 ! print the message !
        jp      ii0

END _in_int
II4:    ARRAY[*BYTE]:='?REDO%R%L%00'
END _inout_c
```

Functions defined in rtc.s module:

```
     cio_init         Assembly routine to initialize the CIO chips w
                      are responsible for Real-Time-Clock (RTC) inte
                      and discrete (bit) I/O to the Digital Avionics
                      interface rack. The three discrete ports on th
                      CIO chip are initialized as follows:
                              PORTA   input_only
                              PORTB   input_only
                              PORTC   input_only
                      The three ports on the second CIO chip are ini
                      as follows:
                              PORTA   output_only
                              PORTB   output_only
                              PORTC   output_only
                      (see the CIO chip manual for more information)

                      ENTRY: none
                       EXIT: none


     rtc_init         Assembly language routine which is installed a
                      interrupt vector for RTC interrupts. The routi
                      calls the head_tracking_stabilization loop and
                      returns. This routine is installed in the PSAP
                      the Z8000 as an optional function of CIO_INIT
                      ENTRY:  none (interrupt)
                       EXIT:  none


     cio_ina          Assembly language routines to input-from/outpu
     cio_outa         the CIO chips. All cio_inx routines acquire in
     cio_inb          from the first CIO chip, port x. All cio_outx
     cio_outb         routines send output to the second CIO chip
     cio_outc

                      ENTRY:  RL7 = data to output (for output, or
                                    none, for input)
                       EXIT:  RL2 = none (for output, or
                                    8-bit data, for input)

Externals  { mess_c,out_char,out_int,out_crlf,head,Cycle_Cntr }
```

```
_cio_control MODULE

CONSTANT

CIO1      :=      %fd01    ! base address of CIO-1 !
MICR1     :=      %fd01    ! master interrupt control register !
MCCR1     :=      CIO1+2   ! master configuration control register !
CTCSR11 :=        CIO1+20 ! Counter/Timer-1 command and status register
CTCSR12 :=        CIO1+22 !    "      "   2     "     "      "       "
CTMSR11 :=        CIO1+56 ! Counter/Timer-1 mode specification register
CTMSR12 :=        CIO1+58 ! Counter/Timer-2 mode specification register
CTCCR1L1 :=       CIO1+34 ! Counter/Timer-1 current count LSB !
CTCCR1L2 :=       CIO1+38 ! Counter/Timer-2 current count LSB !
CTCCR1H1 :=       CIO1+32 ! Counter/Timer-1 current count MSB !
CTCCR1H2 :=       CIO1+36 ! Counter/Timer-2 current count MSB !
CTTCR1L1 :=       CIO1+46 ! Counter/Timer-1 time constant register LSB !
CTTCR1H1 :=       CIO1+44 ! Counter/Timer-1 time constant register MSB !
CTTCR1L2 :=       CIO1+50 ! Counter/Timer-2 time constant register LSB !
CTTCR1H2 :=       CIO1+48 ! Counter/Timer-2 time constant register MSB !
CTIVR     :=      CIO1+8   ! Counter/Timer-1 interrupt vector register !


 ! CIO REGISTER ADDRESSES !

IVR1A     :=      %FD05    ! cio 1, port A, interrupt vector register !
IVR1B     :=      %FD07    ! cio 1, port B, interrupt vector register !
IVR1CT    :=      %FD09    ! cio 1, counter/timer interrupt vector regist
DPPR1C    :=      %FD0B    ! cio 1, port C, data path polarity register !
DDR1C     :=      %FD0D    ! cio 1, port C, data direction register !
SIOCR1C :=        %FD0F    ! cio 1, port C, special I/O register !
PCSR1A    :=      %FD11    ! cio 1, port A, command and status register !
PCSR1B    :=      %FD13    ! cio 1, port B, command and status register !
PDR1A     :=      %FD1B    ! cio 1, port A, data register !
PDR1B     :=      %FD1D    ! cio 1, port B, data register !
PCDR1     :=      %FD1F    ! cio 1, port C, data register !
CVR1      :=      %FD3F    ! cio 1, counter/timer current vector !
PMSR1A    :=      %FD41    ! cio 1, port A, mode specification register !
PHSR1A    :=      %FD43    ! cio 1, port A, handshake specification regis
DPPR1A    :=      %FD45    ! cio 1, port A, data patch polarity register
DDR1A     :=      %FD47    ! cio 1, port A, data direction register !
SIOCR1A :=        %FD49    ! cio 1, port A, special I/O register !
PMSR1B    :=      %FD51    ! cio 1, port B, mode specification register !
PHSR1B    :=      %FD53    ! cio 1, port B, handshake specification regis
DPPR1B    :=      %FD55    ! cio 1, port B, data path polarity register !
DDR1B     :=      %FD57    ! cio 1, port B, data direction register !
SIOCR1B :=        %FD59    ! cio 1, port B, special I/O register !
MICR2     :=      %FE01    ! same as CIO-1 (above) !
MCCR2     :=      %FE03
IVR2A     :=      %FE05
IVR2B     :=      %FE07
IVR2CT    :=      %FE09
DPPR2C    :=      %FE0B
DDR2C     :=      %FE0D
SIOCR2C :=        %FE0F
```

```
PCSR2A  :=      %FE11
PCSR2B  :=      %FE13
PDR2A   :=      %FE1B
PDR2B   :=      %FE1D
PCDR2   :=      %FE1F
CVR2    :=      %FE3F
PMSR2A  :=      %FE41
PHSR2A  :=      %FE43
DPPR2A  :=      %FE45
DDR2A   :=      %FE47
SIOCR2A :=      %FE49
PMSR2B  :=      %FE51
PHSR2B  :=      %FE53
DPPR2B  :=      %FE55
DDR2B   :=      %FE57
SIOCR2B :=      %FE59

EXTERNAL
        _mess_c         PROCEDURE
        _out_char       PROCEDURE
        _out_int        PROCEDURE
        _out_crlf       PROCEDURE
        _head           PROCEDURE
        _Cycle_Cntr     WORD
GLOBAL
        _int_flag       WORD;

GLOBAL
_cio_init PROCEDURE

ENTRY

! on entry r7 contains the desired time constant !

! first initialize the interrupt stuff !

        di      vi,nvi          ! disable vi,nvi !

        ldb     rl0,#%01        ! do a Cio reset !
        outb    MICR1,rl0       ! by setting the reset bit !

        ldctl   r1,psapoff      ! current PSAP  (should be %FD00) !

        add     r1,#%1C         ! offset to FCW location !
        ld      r0,#%4000       ! setup FCW for vectored interrupts !
        ld      @r1,r0          ! store %4000 at xx1c in PSAP !

        add     r1,#02          ! add to to get next address !
        ld      @r1,#_rtc_int   ! set vector to our guy !

        push    @r15,r7         ! save r7 just in case !
        push    @r15,r6         ! save r6 just in case !
        ldb     r10,#%00        ! set MIE (mast. int. en) !
        outb    MICR1,r10       ! put back out !

        ldb     r10,#%82        ! enable sqr wave, continuous cycle!
        outb    CTMSR11,r10     ! put it back now !
        outb    CTMSR12,r10

        ldb     r10,#00         ! zero the interrupt vector !
        outb    CTIVR,r10       ! of CT-3 CIO-1 !
        pop     r6,@r15         ! get time constant for CT-2 !
        outb    CTTCR1L2,r16    ! set up CTTCR1L2 !
        outb    CTTCR1H2,rh6

        pop     r7,@r15         ! set up time constant for CT-1 !
        outb    CTTCR1L1,r17    ! lsh !
        outb    CTTCR1H1,rh7    ! msh !

        ldb     r10,#%c0        ! set IE for Counter/Timer 2 !
        outb    CTCSR12,r10     ! and same for C/T 2 !

        ldb     r10,#%e0        ! clear IE for C/T 1 !
        outb    CTCSR11,r10

        ldb     r10,#%f7        ! enable (timer-1 clocks timer-2), !
                                ! ports A,B, and C of unit 1 !
        outb    MCCR1,r10       ! using MCCR !

! unit 1 initialization !
! set MSR Port A & B to input single buffer !

        ldb     r10,#%10
        outb    PMSR1A,r10
        outb    PMSR1B,r10

! clear PCSR's, DPPR's, and SIOCR's !

        clr     r0
        outb    PCSR1A,r10
        outb    PCSR1B,r10
        outb    DPPR1A,r10
        outb    DPPR1B,r10
        outb    DPPR1C,r10
        outb    SIOCR1A,r10
        outb    SIOCR1B,r10
        outb    SIOCR1C,r10

! set Data Direction for input !

        ldb     r10,#%ff
        outb    DDR1A,r10
        outb    DDR1B,r10
        outb    DDR1C,r10
```

```
! unit 2 initialization !
! clear the reset bit !

        clr     r0
        outb    MICR2,rl0

! set MSR Port A & B to output single buffered !

        ldb     rl0,#%10
        outb    PMSR2A,rl0
        outb    PMSR2B,rl0

! clear PCSR's, DPPR's, and SIOCR's !

        clr     r0
        outb    PCSR2A,rl0
        outb    PCSR2B,rl0
        outb    DPPR2A,rl0
        outb    DPPR2B,rl0
        outb    DPPR2C,rl0
        outb    SIOCR2A,rl0
        outb    SIOCR2B,rl0
        outb    SIOCR2C,rl0

! set Data Direction for output !

        outb    DDR2A,rl0
        outb    DDR2B,rl0
        outb    DDR2C,rl0

! enable ports A, B, & C !

        ldb     rl0,#%94
        outb    MCCR2,rl0

        ldb     rl0,#%80        ! enable master interrupts !
        outb    MICR1,rl0


        ldb     rl0,#%06
        outb    CTCSR11,rl0     ! set trigger and gate bits C/T 1!
        outb    CTCSR12,rl0     ! and C/T 2 !

        ei      vi,nvi
        ret                     ! end of initialization !

END _cio_init
```

```
_rtc_int PROCEDURE

ENTRY

! routine to service the rtc interrupt(s) !

        ldm     regs_save,r0,#15        ! save the context at entry !

        ldb     rl0,#%26                ! clear IUS,IP bits of C/T 2 !
        outb    CTCSR12,rl0

  !      call    _head                    not for debug !

  !      inc     _Cycle_Cntr              not for debug !

        ldm     r0,regs_save,#15        ! restore context !

        iret                            ! return from interrupt !

END _rtc_int

regs_save:      ARRAY   [16 WORD]


_cio_ina PROCEDURE
ENTRY
! input port A !
        inb     rl2,PDR1A
        ret
END _cio_ina


_cio_outa PROCEDURE
ENTRY
! output port A !
        outb    PDR2A,rl7
        ret
END _cio_outa


_cio_inb PROCEDURE
ENTRY
! input port B !
        inb     rl2,PDR1B
        ret
END _cio_inb
```

```
_cio_outb PROCEDURE
ENTRY
! output port B !
        outb    PDR2B,rl7
        ret
END _cio_outb

_cio_outc PROCEDURE
ENTRY
! output port C !
        outb    PCDR2,rl7
        ret
END _cio_outc

END _cio_control
```

```
Functions defined in ad_da.s

      ad_in_a         Assembly language subroutine to acquire 12 bit
                      data from first set of 12 A/D channels. All 12
                      channels are converted.
                      ENTRY:  R7 = address of data buffer (must cont
                                   at least 24 bytes of free space)
                              R6 = 1 if first call, 0 if not
                       EXIT:  none

      ad_in_b         (Same as ad_in_a, except for channels  12-23)

      da_out          Assembly language subroutine to output 12-bit
                      data to the D/A converter.
                      ENTRY:  R7 = address of data buffer
                              R6 = Start channel (0-15)
                              R5 = Number of channels (0-15)
                       EXIT:  none

  Externals    ( cio_outc,cio_outb,cio_outc )


_ad_da MODULE

CONSTANT
 ! A_D INPUT BASE ADDRESS !

BASE1    :=     %f600
BASE2    :=     %f610
CSRL1    :=     BASE1
SCRL1    :=     BASE1+2
MARL1    :=     BASE1+4
DREG1    :=     BASE1+6
CSRL2    :=     BASE2
SCRL2    :=     BASE2+2
MARL2    :=     BASE2+4
DREG2    :=     BASE2+6
RDY      :=     15              ! because of byte swap !
PND      :=     2               ! when doing word I-O !
CAT      :=     1
FST      :=     0

 ! D_A OUTPUT CHANNEL BASE ADDRESS !

BASE     :=     %f710
CHAN0    :=     BASE
CHAN1    :=     BASE+2
CHAN2    :=     BASE+4
CHAN3    :=     BASE+6
CHAN4    :=     BASE+8
CHAN5    :=     BASE+10
```

```
CHAN6    :=      BASE+12
CHAN7    :=      BASE+14
BASE3    :=      %f720
CHAN8    :=      BASE3
CHAN9    :=      BASE3+2
CHAN10   :=      BASE3+4
CHAN11   :=      BASE3+6
CHAN12   :=      BASE3+8
CHAN13   :=      BASE3+10
CHAN14   :=      BASE3+12
CHAN15   :=      BASE3+14

EXTERNAL
        _cio_outc        PROCEDURE
        _cio_outb        PROCEDURE
        _cio_outa        PROCEDURE

GLOBAL

_ad_in_a PROCEDURE
ENTRY
! r7 has the load address for data (passed from C) !
! r6 has: 1 if start/stop channels are to be updated !
!         0 if start/stop channels have already been initialized !
! check for first time thru !
        ld      r4,r6                   ! init bit !
        cp      r4,#1                   ! 1 means re-init !
        jr      z,ad07                  ! go if init desired !
        in      r1,CSRL1
        bit     r1,#PND
        jr      z,ad37
ad07:   ld      r0,#%0010               ! output init bit to control !
        out     CSRL1,r0
ad10:   in      r1,CSRL1                !wait for CAT to clear !
        bit     r1,#CAT
        jr      nz,ad10
        ld      r0,#%0100               ! write 1 to scan !
        out     SCRL1,r0
ad20:   in      r1,CSRL1                ! wait for CAT to clear !
        bit     r1,#CAT
        jr      nz,ad20
        ld      r0,#%0000               ! write 0 to scan !
        out     SCRL1,r0
ad25:   in      r1,CSRL1                ! wait for CAT to clear !
        bit     r1,#CAT
        jr      nz,ad25
        ld      r1,#%0800               ! set SCN bit in control !
        out     CSRL1,r1
        ld      r1,#%0072               ! set pacer to 100KH !
        out     MARL1,r1
ad30:   in      r1,CSRL1                ! wait for scan to clear !
        bit     r1,#CAT
        jr      nz,ad30
        ld      r6,#%0b                 ! do all 12 channels !
        out     SCRL1,r6
ad35:   in      r1,CSRL1                ! wait for CAT to clear !
        bit     r1,#CAT
        jr      nz,ad35
ad37:   ld      r1,#%0900               ! set SCN and software trigger
        out     CSRL1,r1
ad40:   in      r1,CSRL1                ! wait for CAT to clear !
        bit     r1,#C   r  z,a! netiocodad4  ld     r2,#%0C
        ld      r3,r7                   ! start address from C !
ad401:  in      r1,CSRL1
        bit     r1,#RDY
        jr      z,ad401
        in      r0,DREG1
        exb     rh0,rl0
        ld      @r3,r0
        inc     r3,#2
        djnz    r2,ad401                ! loop till all data read !
        ret
! end of new section !

END _ad_in_a


_ad_in_b PROCEDURE
ENTRY
! check for first time thru !
        ld      r4,r6                   ! init bit !
        cp      r4,#1                   ! see if re-init desired !
        jr      z,ad4f
        in      r1,CSRL2
        bit     r1,#PND
        jr      z,ad77
ad4f:   ld      r0,#%0010               ! output INIT bit to control !
        out     CSRL2,r0
ad50:   in      r1,CSRL2                ! wait for CAT to clear !
        bit     r1,#CAT
        jr      nz,ad50
        ld      r0,#%0100               ! write 1 to scan !
        out     SCRL2,r0
ad60:   in      r1,CSRL2                ! wait for CAT to clear !
        bit     r1,#CAT
        jr      nz,ad60
        ld      r0,#%0000               ! write 0 to scan !
        out     SCRL2,r0
ad65:   in      r1,CSRL2                ! wait for CAT to clear !
        bit     r1,#CAT
        jr      nz,ad65
        ld      r0,#%0800               ! set SCN bit in control !
        out     CSRL2,r0
        ld      r1,#%0072               ! set pacer to 100KH !
        out     MARL2,r1
ad70:   in      r1,CSRL2                ! wait for CAT to clear !
        bit     r1,#CAT
        jr      nz,ad70
        ld      r6,#%0b
        out     SCRL2,r6
ad75:   in      r1,CSRL2                   ! wait for CAT to clear !
        bit     r1,#CAT
        jr      nz,ad75
ad77:   ld      r1,#%0900                  ! set SCN and software trigger
        out     CSRL2,r1
ad80:   in      r1,CSRL2                   ! wait for CAT to clear !
        bit     r1,#CAT
        jr      nz,ad80
! new section !
ad800:  ld      r2,#%0c
        ld      r3,r7
ad801:  in      r1,CSRL2
        bit     r1,#RDY
        jr      z,ad801
        in      r0,DREG2
        exb     rh0,rl0
        ld      @r3,r0
        inc     r3,#2
        djnz    r2,ad801
        ret                                ! end of routine !
END _ad_in_b


_da_out PROCEDURE
ENTRY
! r7 has the data address, r6 has the start channel, r5 has the # chan
daloop: cp      r5,#0           ! see if done !
        ret     z               ! return if done !
        ld      r3,r6           ! next channel number !
        inc     r6,#1
        ld      r2,@r7          ! get next data !
        inc     r7,#2           ! point to next data !
        calr    outda           ! output one word !
        dec     r5,#1           ! decrement channel counter !
        jr      daloop          ! loop till done !

outda:  cp      r3,#3
        jr      gt,doit
        push    @r15,r7         ! save r7 for now !
        ld      r7,#2           ! set write enable of 74LS138 !
        call    _cio_outc
        ld      r7,r3           ! output channel number to cio !
        and     r7,#3           ! only allow channels 0-3 !
        call    _cio_outb
        ld      r7,#3           ! disable 74LS138 !
        call    _cio_outc
        pop     r7,@r15
doit:   sll     r3,#1           ! multiply channel by 2 !
        ld      r4,#%f710       ! start address of D/A channels !
        add     r4,r3           ! add offset to start address !
        exb     rh2,rl2         ! exchange bytes for output !
        out     @r4,r2          ! output the word to the Dig-Analog co
        ret                     ! end of the routine !
END _da_out

END _ad_da
```

```
Functions defined in fp_subs.s module:
          All of the below subroutines are used to manipulate
          floating point data.

    +-------------------------------------------------------
    |     "C"-interface routines
    |
    |     The following routines are directly callable from "C".
    |     A brief summary of the function and an example syntax
    |     is given below.
    |
    +-------------------------------------------------------

    fadd    add two floating_point numbers
            ex:     fp_result = fadd(fp1,fp2);

    fmul    multiply two floating point numbers
            ex:     fp_result = fmul(fp1,fp2);

    fdiv    divide two floating point numbers (fp1/fp2)
            ex:     fp_result = fdiv(fp1,fp2);

    fcmp    compare two floating point numbers and returns a longw
            result:
                    comparison              returns
                    ----------              -------
                    fp1 < fp2               -1
                    fp1 = fp2                0
                    fp1 > fp2                1

            ex:     long_result = fcmp(fp1,fp2);

    fint    extracts the integer portion of a floating point numbe
            returns the result as a longword integer.
            ex:     long_result = fint(fp1);

    frac    extracts the fractional portion of a floating point nu
            ex:     fp_result = frac(fp1);

    fsub    subracts two floating point numbers.
            ex:     fp_result = fsub(fp1,fp2);

    fp_in   allows keyboard input of floating point number. The ro
            accepts standard forms for input (including exponentia
            ex:     fp_result = fp_in();

    fp_out  displays a floating point number in non-exponential fo
            ex:     fp_out(fp1);

    fp_out_e displays a floating point number in exponential forma

    fpconv  converts a long_integer to floating point representati
            ex:     fp_result = fpcon(long_int);
```

```
fpcon   obtains the floating point equivalent of a character s
        expression.
        ex:     fp_result = fpcon("-10.256E-3");


+-------------------------------------------------------------
|       Assembly-interface routines
|
|       The following routines are not directly callable from
|       "C". A brief summary of the function is given below.
|
|       In each case:
|          ENTRY:
|               RR4 = floating point argument 1
|               RR2 = floating point argument 2 (optional)
|
|          EXIT:
|               RR2 = resulting number (may be floating point
|                         longword_integer
+-------------------------------------------------------------

fp_add          add             RR2 = RR4+RR2           (fp re
fp_mult         multiply        RR2 = RR2*RR4           (fp re
fp_div          divide          RR2 = RR2/RR4           (fp re
fp_frac         fraction        RR2 = frac(RR2)         (fp re
fp_cmp          compare         RR2 = sgn(RR2-RR4)      (long
fp_int          integer         RR2 = int(RR2)          (long


+-------------------------------------------------------------
|       Assembly-utility routines
|
|       The following routines are not directly callable from
|       "C". A brief summary of the function is given below.
|
+-------------------------------------------------------------


split           separates the "sign","exponent", and "mantissa
mul_mant        multiplies two mantissas together
div_mant        divides two mantissas
unsplit         recombines the "sign","exponent", and "mantiss

Externals  { out_dec,out_char,in_string,out_crlf,mess_c,out_int }
```

```
fp_subs MODULE

EXTERNAL
        _out_dec        procedure
        _out_char       procedure
        _in_string      procedure
        _out_crlf       procedure
        _mess_c         procedure
        _out_int        procedure

GLOBAL

_fadd PROCEDURE

ENTRY
        ! C callable procedure to perform floating point addition of !
        ! two numbers (must be floating point) and return the result !

        ! ENTRY:
        !       rr6 = addend
        !       rr4 = additive
        !
        !  EXIT:
        !       rr2 = addend + additive

        ldl     rr2,rr6                 ! substitute rr2 for rr6 !
        call    fp_add                  ! which is used in fp_add !
        ret                             ! already in rr2 !
END _fadd



_fmul PROCEDURE

ENTRY

        ! C callable procedure to perform floating point multiplicatio
        ! of two numbers (must be floating point) and return the resul

        ! ENTRY:
        !       rr6 = multiplicand
        !       rr4 = multiplier

        !  EXIT:
        !       rr2 = multiplicand * multiplier

        ldl     rr2,rr6                 ! rr2 for fp_mult !
        call    fp_mult
        ret
END _fmul
```

```
_fdiv PROCEDURE

ENTRY
        ! C callable procedure to perform floating point division !
        ! of two numbers (must be floating point) and return the resul

        ! ENTRY:
                rr6 = dividend
                rr4 = divisor

          EXIT:
                rr2 = dividend/divisor

        ldl     rr2,rr6
        call    fp_div
        ret
END _fdiv


_fcmp PROCEDURE

ENTRY

        ! C callable procedure to compare two floating point numbers !
        ! A and B and return either a -1, 0, or a 1 to represent:    !
        !   A<B, A=B, and A>B, respectively                          !

        ! ENTRY:
                rr6 = A
                rr4 = B

          EXIT:
                rr2 = -1,0,1

        ldl     rr2,rr6
        call    fp_cmp
        ret
END _fcmp
```

```
_fint PROCEDURE

ENTRY
        ! C callable routine to return a LONG INTeger representing !
        ! the integer part of a floating point number !

        ! ENTRY:
                rr6 = floating point number of return integer part of

          EXIT:
                rr2 = integer part of rr6 (not in floating point)    !

        ldl     rr2,rr6
        call    fp_int
        ret
END _fint


_frac PROCEDURE

ENTRY
        ! C callable procedure to return the fractional part of a !
        ! floating point number as a floating point number !

        ! ENTRY:
                rr6 = floating point number (input)

          EXIT:
                rr2 = fractional part of input (floating point format)

        ldl     rr2,rr6
        call    fp_frac
        ret
END _frac
```

```
_fsub PROCEDURE

ENTRY
        ! C callable procedure to perform subtraction on two floating
        ! point numbers !

        ! ENTRY:
                rr6 = subtrahend
                rr4 = subtractor

          EXIT:
                rr2 = subtrahend - subtractor

        xor     r4,#%8000               ! make subtractor negative !
        ldl     rr2,rr6

        call    fp_add                  ! and add to negated subtracto

        ret                             ! the end !
END _fsub



fp_add PROCEDURE

ENTRY

! rr2 contains bit encoded data to add to rr4 !
! this routine destroys all other register data and returns the
  result in rr2 !

        ldl     arg1,rr2                ! save addend !
        ldl     arg2,rr4                ! save additive !

        cpl     rr2,#0                  ! if addend is zero, then !
        jr      nz,fpad1                ! return additive, else jump !
        ldl     rr2,rr4                 ! result is additive if zero !
        ret                             ! done !

fpad1:
        cpl     rr4,#0                  ! if additive is zero, then !
        jr      nz,fpad2                ! return addend, else jump !
        ret                             ! additive was zero, so result
                                        ! is already in rr2 !

fpad2:
        call    split

        testl   arg1                    ! test for negative mant !
        jr      pl,fpad3                ! go if positive !
        ldl     rr6,#0                  ! make negative by subtraction
        subl    rr6,rr10                ! and move result back into rr
        ldl     rr10,rr6                ! done !
```

```
fpad3:
        testl   arg2                    ! same as for addend !
        jr      pl,fpad4
        ldl     rr6,#0
        subl    rr6,rr8
        ldl     rr8,rr6
fpad4:
        cp      r12,r13                 ! while (aexp <> bexp ) ... !
        jr      z,fpad7                 ! go if equal  !

        jr      gt,fpad5                ! go if aexp > bexp !
        inc     r12,#1                  ! increment aexp !
        sral    rr10,#1                 ! divide amant by 2 !
        jr      fpad6                   ! skip next part !
fpad5:
        inc     r13,#1                  ! increment bexp !
        sral    rr8,#1                  ! bexp = bexp/2 !
fpad6:
        jr      fpad4                   ! end while.... !
fpad7:
        cpl     rr10,#0                 ! if one was shifted out !
        jr      nz,fpad8                ! go if amant is not zero !
        ldl     rr2,arg2
        ret
fpad8:
        cpl     rr8,#0                  ! see if other shifted out !
        jr      nz,fpad9
        ldl     rr2,arg1
        ret
fpad9:
        addl    rr8,rr10                ! add mantissas !
        ! now, resulting mantissa is in rr8, resulting exponent in r13

        ld      sign,#0                 ! zero negative sign flag !

        testl   rr8                     ! see if mantissa negative !
        jr      pl,fpad10               ! go if positive !
        ldl     rr6,#0                  ! else make positive !
        subl    rr6,rr8                 ! and put back into rr8 !
        ldl     rr8,rr6                 ! done !
        ld      sign,#%ff               ! set negative sign flag !

        ld      r12,sign                ! put sign in r12 for unsplit
        call    unsplit

        ret
END fp_add
```

```
arg1:
        wval    0
        wval    0
arg2:
        wval    0
        wval    0
sign:
        wval    0


split PROCEDURE

ENTRY

! rr2 is arg 1, rr4 is arg 2 !

! results: r12 = exp(arg 1)    from RR2
           r13 = exp(arg 2)    from RR4
           rr10= mant(arg 1)   from RR2
           rr8 = mant(arg 2)   from RR4
           arg1 = rr2
           arg2 = rr4 !

        and     r2,#%7f00               ! mask out exponent of arg1!
        exb     rl2,rh2                 ! place into lower half !
        sub     r2,#32                  ! subtract off bias !
        ld      r12,r2                  ! save in r12 !

        and     r4,#%7f00               ! mask out exponent of arg2 !
        exb     rl4,rh4                 ! place into lower half !
        sub     r4,#32                  ! subtract off bias !
        ld      r13,r4                  ! save in r13 !

        ldl     rr2,arg1                ! get uncorrupted arg1 back !
        and     r2,#%ff                 ! only need lower 8 bits !
        or      r2,#%100                ! set hidden bit !
        ldl     rr10,rr2                ! save arg2 mant in rr10 !

        ldl     rr4,arg2                ! same as above !
        and     r4,#%ff
        or      r4,#%100
        ldl     rr8,rr4                 ! save additive mant in rr8 !

        ret
END split
```

```
GLOBAL

fp_mult PROCEDURE

ENTRY

! rr2 contains bit encoded data to multiply !
! by rr4. This routine destroys all other register !
! data and returns the answer in rr2. !

        ldl     arg1,rr2                ! save multiplicand !
        ldl     arg2,rr4                ! save multiplier !

        cpl     rr2,#0                  ! if zero then !

        jr      nz,fpmul1               ! return zero, else jump !
        ldl     rr2,#0                  ! result is zero !
        ret                             ! done !

fpmul1:
        cpl     rr4,#0                  ! if zero then !

        jr      nz,fpmul2               ! return zero else jump !
        ldl     rr2,#0                  ! result is zero !
        ret                             ! done !

fpmul2:
        call    split                   ! mask out exp' and mant' !
! r13 is exp and rr10 is mant of multiplicand !
! r12 is exp and rr8  is mant of multiplier !

        cpl     rr8,#0                  ! if zero then !

        jr      nz,fpmul3               ! return zero, else jump !
        ldl     rr2,#0                  ! result is zero !
        ret                             ! done !

fpmul3:
        cpl     rr10,#0                 ! if zero then !

        jr      nz,fpmul4               ! return zero, else jump !
        ldl     rr2,#0                  ! result is zero !
        ret                             ! done !

fpmul4:
        ldl     rr4,rr8                 ! set up to multiply the !
        ldl     rr6,rr10                ! mantissi together !
        call    _mul_mant               ! returns result in rr2 !

        ld      sign,#0                 ! start test for sign !
        testl   arg1                    ! check to see if !
        jr      lt,fpmul5               ! result is positive or !
        testl   arg2                    ! and set sign bit !
        jr      lt,fpmul6               ! accordingly !
        jr      fpmul7                      ! positive !
fpmul5:
        testl   arg2
        jr      gt,fpmul6
        jr      fpmul7                      ! positive !
fpmul6:
        ld      sign,#%ff                   ! ñegative !
fpmul7:

        ldl     rr8,rr2                     ! setup to call unstrip !
                                            ! resulting mantissa in rr8 !
        add     r13,r12                     ! sum of exp in r13 !

        ld      r12,sign                    ! sign in r12, 0 if pos, ff if
        call    unsplit                     ! reforms the product into !
                                            ! a 32 bit fp number !
        ret                                 ! done !

END fp_mult



GLOBAL

_mul_mant PRC:EDURE

ENTRY

        slal    rr6,#6
        slal    rr4,#6
        ldl     rr2,rr4
        multl   rq4,rr2
        sral    rr4,#4
        ldl     rr2,rr4
        ret
END _mul_mant


_div_mant PROCEDURE

ENTRY

        ldl     rr2,rr4
        ldl     rr4,rr6
        ldl     rr6,#0
        divl    rq4,rr2
        ldl     rr2,rr6
        srll    rr2,#8
        ret
END _div_mant
```

```
GLOBAL

unsplit PROCEDURE

ENTRY

! enter this routine with cmantissa in rr8 and cexp in r13 !
! sign in r12 , 0 if pos and ff if neg !
! returns result in rr2 !

fpad10:
        ld      r0,#0                  ! i = 0!
        ldl     rr2,#0
        testl   rr8                    ! already done if zero !
        ret     z
fpad11:
        ldl     rr2,rr8                ! rr2 is cmant !
        and     r2,#%ff00              ! keep only upper bits !
        clr     r3
        cpl     rr2,#%1000000          ! (see if normalized) !
        jr      z,fpad14               ! go if equal to 01000000 !
        cp      r0,#32                 ! make sure r0 not equal to 32
        jr      z,fpad14               ! else leave the loop !
        cpl     rr8,#%1000000          ! see if bigger or smaller !
        jr      lt,fpad12              ! go if smaller !

        sral    rr8,#1                 ! divide mantissa !
        inc     r13                    ! and add one to exponent !
        jr      fpad13                 ! skip next part !
fpad12:
        slal    rr8,#1                 ! multiply mantissa by 2 !
        dec     r13                    ! and subtract one from expone

fpad13:
        inc     r0                     ! add one to r0 !
        jr      fpad11                 ! loop till normalized !
fpad14:
        cp      r13,#-31               ! see if underflow !
        jr      gt,fpad15              ! go if not !
        ldl     rr2,#0                 ! else, underflow error !
        ret
fpad15:
        cp      r13,#31                ! see if overflow !
        jr      lt,fpad17              ! go if not !
        ldl     rr8,#%3effffff         ! else is, and gets max value

        cp      r12,#%ff               ! see if negative !
        jr      nz,fpad16              ! go if positive !
        or      r8,#%8000              ! else, set sign bit !
fpad16:
        ldl     rr2,rr8                ! return with limited result !
        ret
fpad17:
        ldl     rr2,rr8                ! get mantissa into rr2 !
        and     r2,#%00ff               ! clear exponent part !

        cp      r12,#%ff                ! see if negative !
        jr      nz,fpad18               ! go if positive !

        or      r2,#%8000               ! or in sign bit !
fpad18:
        add     r13,#32                 ! add bias to exponent !
        ld      r4,r13
        exb     rh4,rl4
        ld      r13,r4
        or      r2,r13                  ! now, word is complete !
        ret
END unsplit


GLOBAL

fp_div PROCEDURE

ENTRY

! rr2 contains bit encoded data to divide by rr4. !
! this routine destroys all other register data  !
! and returns the answer in rr2 . !

        ex      r3,r5
        ex      r2,r4
        ldl     arg1,rr2                ! save dividend !
        ldl     arg2,rr4                ! save divisor  !

        cpl     rr4,#0                  ! if zero then, !
        jr      nz,fpdiv1               ! return zero, else jump !
        ldl     rr2,#0                  ! result is zero !
        ret                             ! done !

fpdiv1:
        cpl     rr2,#0                  ! if zero then, !
        jr      nz,fpdiv2               ! return max, else jump !
        ldl     rr2,#%3effffff          ! result is max !
        ret                             ! done !

fpdiv2:
        call split                      ! mask out exp' and mant' !

! r13 is exp and rr10 is mant of divisor !
! r12 is exp and rr8  is mant of dividend !

        cpl     rr10,#0                 ! if zero then !
        jr      nz,fpdiv3               ! return max, else jump !
        ldl     rr2,#%3effffff          ! result is max !
        ret                             e !pdi   cpl rr8,#0
        jr      nz,fpdiv4               ! return zero, else jump !
```

```
          ldl      rr2,#0                      ! result is zero !
          ret                                  ! done !

fpdiv4:
          cpl      rr8,rr10                    !for mant div !
          jr       lt,fpdiv5                   ! if dividend => divisor then,
          sral     rr8,#1                      ! divide dividend man by 2 !
          inc      r13,#1                      ! bump up the exp !
          jr       fpdiv4                      ! try till true !

fpdiv5:
          ldl      rr6,rr8                     ! set up to divide the mant !
          ldl      rr4,rr10                    ! by one another !
          call     _div_mant                   ! returns result in rr2 !

          ld       sign,#0                     ! start test for sign !

          testl    arg1                        ! check to see if !
          jr       lt,fpdiv6                   !result is positive !
          testl    arg2                        ! or neg and set sign flag !
          jr       lt,fpdiv7                   ! accordingly !
          jr       fpdiv8

fpdiv6:
          testl    arg2
          jr       gt,fpdiv7
          jr       fpdiv8

fpdiv7:
          ld       sign,#%ff                   ! neg !

fpdiv8:
          ldl      rr8,rr2                     ! set up for unsplit !
          sub      r13,r12                     ! subtract exponents !
          nop
          ld       r12,sign                    ! still setting up for unsplit
          call     unsplit                     ! result in rr2 !
          ret
END fp_div
```

```
fp_int PROCEDURE

ENTRY
! rr2 contains number to return integer part of in rr2 !
        ldl     arg1,rr2                ! save off  argument !
        call    split                   ! resulting mantissa in rr10,

        ld      sign,#0                 ! determine the sign !
        testl   arg1
        jr      pl,fpint1               ! go if positive !
        ld      sign,#%ff               ! else, flag as negative !
fpint1:
        cp      r12,#0                  ! if exp < 0, then int(arg1)=0
        jr      ge,fpint2               ! go if exp > 0 !
        ldl     rr2,#0                  ! else, result is zero !
        ret
fpint2:
        cp      r12,#31                 ! if exp > 31 then overflow !
        jr      le,fpint3               ! go if no overflow !
        ldl     rr2,#%7fffffff          ! rr2 gets max positive value
        cp      sign,#%ff               ! see if sign < 0 !
        ret     nz                      ! done if positive !
        addl    rr2,#1                  ! else make max negative %8000
        ret                             ! and return !
fpint3:
        ldl     rr2,#0                  ! zero accumulator !
fpint4:
        cp      r12,#0                  ! while exp >= 0 !
        jr      lt,fpint6               ! go if exp < 0 !

        addl    rr2,rr2                 ! double accum !
        bit     r10,#8                  ! check bit of mant !
        jr      z,fpint5                ! go if bit is zero !
        addl    rr2,#1                  ! increment accum !
fpint5:
        and     r10,#%ff                ! clear bit !
        addl    rr10,rr10               ! double mant !
        dec     r12,#1                  ! decrement exp !
        jr      fpint4                  ! loop while... !

fpint6:
        cp .    sign,#%ff               ! if sign <> 0 !
        jr      nz,fpint7               ! then, negate result !
        ldl     rr8,#0                  ! use rr8 !
        subl    rr8,rr2                 ! make neg !
        ex      r2,r8
        ex      r3,r9                   ! now result in rr2 !
fpint7:
        ret                             ! end of fp_int !
END fp_int
```

```
fp_frac PROCEDURE

ENTRY
        ! rr2 is fp number input, rr2 is fp fractional output !
        ldl     arg1,rr2
        testl   rr2
        jr      nz,fpfrc1                    ! go if input is non zero !
        ldl     rr2,#0                       ! else, return zero as result
        ret
fpfrc1:
        call    split                        ! rr10 is mant, r12 is exp !
        cp      r12,#-1                      ! if exp <= -1 then already a
        jr      gt,fpfrc2                    ! go if > -1 !
        ldl     rr2,arg1                     ! already fraction!
        ret
fpfrc2:
        cp      r12,#-1                      ! while exp > -1 ... !
        jr      le,fpfrc3                    ! go if exp <= -1 !
        dec     r12,#1                       ! exp = exp -1 !
        addl    rr10,rr10                    ! multiply mant by 2 !
        jr      fpfrc2                       ! end while !

fpfrc3:
        and     r10,#%01ff                   ! mask out stuff > 1 in mant !
        ld      sign,#0
        cp      arg1,#0                      ! see if < 0 !
        jr      ge,fpfrc4                    ! go if >= 0 !
        ld      sign,#%ff                    ! else, flag as < 0 !
fpfrc4:
        ldl     rr8,rr10                     ! setup for unsplit !
        ld      r13,r12                      !  ...             !
        ld      r12,sign                     !    ...           !
        call    unsplit
        ret                                  ! end of fp_frac !
END fp_frac
```

```
_fp_out PROCEDURE

ENTRY
        ldl     rr2,rr6                 ! copy C variable to rr2 !
        ! rr2 has fp number of output !
        ldl     arg1,rr2                ! save off !
        call    split                   ! rr10 has mant, r12 has exp !

        testl   arg1                    ! see if number < 0 !
        jr      ge,fpout1               ! go if >= 0 !
        ldb     rl7,#%2d                ! '-' sign !
        call    _out_char               ! assembly routine !
        ldl     rr2,arg1                ! get rr2 back and clear sign
        res     r2,#15                  ! clear the sign bit !
        ldl     arg1,rr2                ! back in arg1 otay !

fpout1:
        ldl     rr2,arg1                ! get arg1 back :
        call    fp_int                  ! get integer part back !
        pushl   @r15,rr2                ! save on stack !
        ldl     rr2,arg1                ! get arg1 back !
        call    fp_frac                 ! get fractional part back !
        popl    rr6,@r15                ! get integer back into rr6 !
        pushl   @r15,rr2                ! save rr2 !
        call    _out_dec                ! 'C' routine to output intege
        ldb     rl7,#%2e                ! '.' !
        call    _out_char               ! output it to CRT !

        popl    rr2,@r15                ! get rr2 back !

        ld      dig_count,#0            ! digit counter !
fpout2:
        testl   rr2                     ! stop if goes to zero !
        jr      z,fpout3                ! go if fractional goes to zer

        cp      dig_count,#05           ! stop after 6 digits !
        jr      gt,fpout3               ! stop if 6 done already !
        inc     dig_count

        ldl     rr4,#%23400000          ! the number 10 !
        call    fp_mult                 ! result in rr2 !
        pushl   @r15,rr2                ! save result of mult !
        call    fp_int                  ! get integer part (0-9) !
        ldb     rl7,rl3                 ! output as digit !
        addb    rl7,#%30                ! ascii offset !
        call    _out_char               ! output to screen !
        popl    rr2,@r15                ! get rr2 back !
        call    fp_frac                 ! result in rr2 !
        jr      fpout2                  ! loop till done !
fpout3:
        ret                             ! end of fp_out !
END _fp_out
```

```
dig_count:
        wval    0

fp_cmp PROCEDURE

ENTRY
        ! rr2 is compared with rr4, returns -1,0,1 for <,=,> !
        ldl     arg1,rr2                        ! save args !
        ldl     arg2,rr4                        ! ...       !
        call    split                           ! rr10,r12 is mant,exp arg1 !
                                                ! rr8 ,r13 is mant,exp arg2  !
        ldl     rr2,arg1                        ! get args back !
        ldl     rr4,arg2                        ! done !
        and     r2,#%8000                       ! mask sign bit !
        and     r4,#%8000                       ! of both arguments !
        sub     r2,r4                           ! sgn(arg1) - sgn(arg2) !
        jr      eq,fpcmp1                       ! go if same sign !
        jr      gt,fpcmp3                       ! arg2 < arg1 !
        jr      fpcmp2                          ! arg1 < arg2 !
fpcmp1:
        cp      r12,r13                         ! compare the exponents !
        jr      eq,fpcmp4                       ! go if equal !
        jr      gt,fpcmp6                       ! go if exp(arg1) > exp(arg2)
        jr      fpcmp7                          ! go if arg1 < arg2 !
fpcmp2:
        ldl     rr2,#-1                         ! arg1 < arg2 !
        ret
fpcmp3:
        ldl     rr2,#1                          ! arg1 > arg2 !
        ret
fpcmp4:
        subl    rr10,rr8                        ! exponents same, test mantiss
        jr      eq,fpcmp5                       ! equal !
        jr      gt,fpcmp6                       ! arg1 > arg2 !
        jr      fpcmp7                          ! arg1 < arg2 !
fpcmp5:
        ldl     rr2,#0                          ! equal !
        ret
fpcmp6:
        ldl     rr2,#1
        jr      fpcmp8
fpcmp7:
        ldl     rr2,#-1
fpcmp8:
        testl   arg1                            ! if args < 0 , complement res
        ret     gt
        testl   rr2                             ! ok if arg is zero !
        ret     z
        jr      gt,fpcmp2                       ! else change 1 to -1, visa-ve
        jr      fpcmp3

END fp_cmp
```

```
_fp_in PROCEDURE

ENTRY

        ! gets characters into array, converts to real in rr2 !
        ld      r7,#string              ! address of string storage !
        ld      r6,#20                  ! num characters !
        call    _in_string              ! get chars !
fp_in_con:
        ld      mult_factor,#%2000      ! 1.0 !
        ld      mult_factor+2,#%0000

        ld      point,#0                ! dec point not encountered !
        ld      ntor,#0                 ! number of digits to right of
        ldl     rr2,#0                  ! zero result !
        ld      r14,#string             ! string pointer !
        ld      in_sign,#0              ! sign is positive !

fpin1:
        testb   @r14                    ! while still more chars !
        jp      z,fpin7                 ! go if done !
        ldb     rl0,@r14                ! get next character !
        cpb     rl0,#%2e                ! see if '.' !
        jp      nz,fpin2                ! go if not !
        ld      point,#1                ! else flag point !
fpin2:
        cpb     rl0,#'-'                ! see if '-' !
        jp      nz,fpin25               ! go if not a '-' !
        ld      in_sign,#1              ! else, flag as < 0 !
fpin25:
        cpb     rl0,#'E'                ! see if exponent next !
        jr      nz,fpin3                ! go if not !
        jp      fpin8                   ! else, it is, so process !
fpin3:
        cpb     rl0,#'9'                ! make sure a valid character
        jp      gt,fpin45               ! go if too big !
        cpb     rl0,#'0'
        jp      lt,fpin45               ! go if too small !

        test    point                   ! see if to right of '.' !
        jp      z,fpin4                 ! go if not to right yet !
fpin4:
        test    point                   ! don't mult by 10 if '.' foun
        jp      nz,fpin40               ! go if '.' already found !
        push    @r15,r0                 ! save character !
        ldl     rr4,#%23400000          ! constant : 10 !
        call    fp_mult                 ! 10 * result !
        pop     r0,@r15                 ! get character back !

fpin40:
        subb    rl0,#%30                ! subtract ascii bias !
        ldb     rh0,#0                  ! zero upper half or r0 !
        add     r0,r0                   ! double result !
                                        ! for indexing into contab !
        add     r0,r0                   ! double again (4 bytes each)
        ld      r1,r0                   ! transfer to r1 for indexing
        ldl     rr4,contab(r1)          ! get constant !

! now, rr2 is the running result
        rr4 is the digit just typed in
!
        pushl   @r15,rr2                ! save rr2 temporarily !

! save running result !

        ldl     rr2,mult_factor         ! get mutiplication factor !
        test    point                   ! if dp found, divide by 10 an
        jr      z,fpin41                ! go if no point found yet !

        pushl   @r15,rr4                ! else, save rr4 !
        ldl     rr4,#%23400000          ! divide mult_factor by 10 !
        call    fp_div                  ! do the divide !
        ldl     mult_factor,rr2         ! new one stored off !
        popl    rr4,@r15                ! get back digit constant !

        ldl     rr2,mult_factor         ! get new mult_factor !
fpin41:

        call    fp_mult                 ! multiply digit const by mult

        popl    rr4,@r15                ! get what was rr2 back (resul

        call    fp_add                  ! add result to digit*mult_fac

fpin45:
        inc     r14                     ! bump character pointer !
        jp      fpinl                   ! loop till done !

fpin7:
        test    in_sign                 ! see if < 0 !
        ret     z                       ! finished if not < 0 !

        or      r2,#%8000               ! else, set sign bit !
        ret                             ! before returning !
fpin8:
        ! here if 'E' was found in input, indicating exponential notat
        pushl   @r15,rr2                ! save base10 mantissa on stac
        inc     r14                     ! but first bump char pointer
        ld      r2,in_sign              ! get in_sign !
        ld      r3,point                ! and point !
        pushl   @r15,rr2                ! and save on stack !
        ldl     rr2,mult_factor         ! get mult_factor !
        pushl   @r15,rr2                ! and save on stack !
        ld      in_sign,#0              ! zero out in_sign and point !
        ld      point,#0                ! so exponent value will be co
        ldl     rr2,#%20000000          ! back to original mult_factor
        ldl     mult_factor,rr2
        ldl     rr2,#0                  ! zero out accumulator !
        call    fpin1                   ! this is recursive !
        ldl     rr6,rr2                 ! only the integer part counts
        call    _fint                   ! result in rr2 (should be onl
        ldl     rr4,rr2                 ! put result into rr4 !

        popl    rr2,@r15                ! mult_factor is back into rr2
        ldl     mult_factor,rr2         ! store back off !

        popl    rr2,@r15                ! r2 = in_sign, r3 = point !
        ld      in_sign,r2
        ld      point,r3

        popl    rr2,@r15                ! rr2 has mantissa again !

fpin9:  testl   rr4                     ! see if exponent gone to zero
        jr      z,fpin7                 ! go if done !

        jr      gt,fpin10               ! go if exp > 0 !

        addl    rr4,#1                  ! increment exponent !
        pushl   @r15,rr4                ! save exponent on stack !
        ldl     rr4,#%23400000          ! divide result by 10 !
        call    fp_div                  ! after adjusting exponent upw
        popl    rr4,@r15
        jr      fpin9                   ! and loop !

fpin10:
        subl    rr4,#1                  ! decrement exponent !
        pushl   @r15,rr4                ! save exponent on stack !
        ldl     rr4,#%23400000          ! multiply by 10 !
        call    fp_mult                 ! after adjusting exponent dow
        popl    rr4,@r15
        jr      fpin9                   ! and loop !

END _fp_in
```

```
point:
        wval    0
ntor:
        wval    0
in_sign:
        wval    0
string:
        array   [25 word]
spaces:
        array   [*byte] := '  '
        wval    %0000
save1:
        array   [20 word]
contab:
        wval    %0000                    ! 0.0 !
        wval    %0000
        wval    %2000                    ! 1.0 !
        wval    %0000
        wval    %2100                    ! 2.0 !
        wval    %0000
        wval    %2180                    ! 3.0 !
        wval    %0000
        wval    %2200                    ! 4.0 !
        wval    %0000
        wval    %2240                    ! 5.0 !
        wval    %0000
        wval    %2280                    ! 6.0 !
        wval    %0000
        wval    %22c0                    ! 7.0 !
        wval    %0000
        wval    %2300                    ! 8.0 !
        wval    %0000
        wval    %2320                    ! 9.0 !
        wval    %0000
mult_factor:
        wval    %0000
        wval    %0000
```

```
_fpcon PROCEDURE

ENTRY

        ! C callable procedure to convert a character constant !
        ! floating point number into an actual floating point number !
        ! usage:
        !       fp_number = fpcon("3.14159");

        ld      r4,#string              ! address of string area !
fpcon1:
        ldb     rl0,@r7                 ! copy characters into string
                                        ! area used by fpin !
        ldb     @r4,rl0                 ! move one !
        inc     r7
        inc     r4
        cpb     rl0,#0                  ! see if done !
        jr      nz,fpcon1               ! loop till done (0 found) !

        call    fp_in_con               ! go do it !
        ret                             ! the end !
END _fpcon



_fp_out_e PROCEDURE

ENTRY
 ! procedure to printout the variable in rr6 in exponential (base 10)

        ld      exp10,#0                ! base ten exponent is zero fi

        ldl     rr2,#0
        cpl     rr6,#0
        jr      z,fpoe1                 ! go if argument is a zero !

        ldl     temp1,rr6               ! save argument !
        res     r6,#15                  ! clear sign bit for compare !
        ldl     rr4,#%20000000          ! constant 1.00 !
        call    _fcmp                   ! rr2 = -1,0,1 if <1,=1,>1 !
        testl   rr2                     ! check sign of comparison !
        jp      lt,fpoe2                ! go if < 1 !

        ldl     rr6,temp1               ! see if > 9 !
        res     r6,#15
        ldl     rr4,#%23200000          ! constant 9.00 !
        call    _fcmp                   ! same return status as above
        testl   rr2                     ! check sign of comparison !
        jp      gt,fpoe3                ! go if > 9 !

        ldl     rr2,temp1
```

```
fpoel:
        ldl     rr6,rr2                 ! print mantissa part !
        call    _fp_out
        ldb     rl7,#'E'                ! print 'E' !
        call    _out_char
        ld      r7,exp10                ! load exp10 into rr6 !
        exts    rr6                     ! extend sign !
        test    r6                      ! see if >= 0 !
        jr      lt,fpoell               ! go if < 0 !
        pushl   @r15,rr6
        ldb     rl7,#'+'                ! else, print '+' !
        call    _out_char
        popl    rr6,@r15
fpoell:
        call    _out_dec                ! output the exponent !
        ldl     rr2,#1                  ! return 1 !
        ret
fpoe2:
        ! rr7 < 1.0 !
        ldl     rr2,templ               ! get argument !
fpoe22:
        pushl   @r15,rr2                ! save result !
        ldl     rr6,rr2                 ! compare with 1.0 again !
        res     r6,#15                  ! clear sign bit !
        ldl     rr4,#%20000000          ! 1.0 constant !
        call    _fcmp
        testl   rr2                     ! check sign of result !
        popl    rr2,@r15                ! get result back !
        jp      ge,fpoel                ! already ready for output !

        ldl     rr4,#%23400000          ! multiply by 10 !
        call    fp_mult
        dec     exp10                   ! and decrement exponent !
        jr      fpoe22                  ! loop till done !
fpoe3:
        ! rr7 > 9.0 !
        ldl     rr2,templ
fpoe33:
        pushl   @r15,rr2                ! save result !
        ldl     rr6,rr2
        res     r6,#15
        ldl     rr4,#%23200000          ! 9.0 constant !
        call    _fcmp
        testl   rr2
        popl    rr2,@r15
        jp      le,fpoel

        ldl     rr4,#%23400000          ! divide by 10.0 !
        call    fp_div
        inc     exp10
        jr      fpoe33                  ! loop till ready to print !
END _fp_out_
templ:  wval    %0000
        wval    %0000
```

```
exp10:  wval    %0000
        wval    %0000


_fpconv PROCEDURE

ENTRY
        ! converts long integer in rr6 to fp number in rr2 !

        ldl     rr8,rr6                         ! setup for call to unsplit !
        ld      r12,#0                          ! make positive, default !
        testl   rr8                             ! if negative, make pos, set s
        jr      ge,fpconvl                      ! go if pos !
        ldl     rr2,#0
        subl    rr2,rr8                         ! else, make positive !
        ldl     rr8,rr2
        ld      r12,#%ff                        ! and flag as negative !
fpconvl:
        ld      r13,#24                         ! original exponent is 12 !
        jp      unsplit                         ! normalize and return !

END _fpconv


END fp_subs
```

APPENDIX B

ADDISP

```
/* program to input and display data from the A/D board */
/* external references */

extern  int  in_char(),in_int(),wait_char(),out_int(),mess_c();
extern  int  ad_in_a(), ad_in_b();
        int  ch_array[8],count2;
        int  adin[30],*pointer;

/* local variables */
        int  num_chans,count;
        char con_dis,c,cntrl;

main ()
{
        out_crlf();
        mess_c("A/D input and display program ");
        out_clrf();
        out_clrf();
        mess_c("Display how many channels (0-7) :");
        num_chans = in_int();
        if (num_chans != 0)  {
                if (num_chans > 7) num_chans = 7;
                out_clrf();
                mess_c("Number of channels set to:");
                out_int(num_chans);
                out_crlf();
                for (count=1;count<=num_chans;count++) {
                        mess_c("column");
                        out_int(count);
                        mess_c("  ");
                        ch_array[count]=in_int();
                        out_clrf();
                        } /* for */
                out_clrf();
                mess_c("Continuout or Discrete sampling (C/D) :");
                con_dis=wait_char();
                if ((con_dis !='C') & (con_dis != 'D')) con_dis = 'D';
                out_clrf;
                mess_c("Sampling set to :");
                out_char(con_dis);
                out_clrf();
                /* the following is the main routine */

                c=count2=0;
                while (((cntrl=in_char()) != 0X1B & c != 0X1B)) {
                    if ((cntrl=='C') | (c=='C')) con_dis ='C';
                    if ((cntrl=='D') | (c=='D')) con_dis ='D';
                    if (count2==15) {  /* redisplay channel numbers */
                         out_clrf();
                         for (count=1;count<=num_chans;count++) {
                         mess_c("chan:");
                         out_int(ch_array[count]);
                         mess_c("  ");
                         } /* for */
```

```
                       out_clrf();
                       for (count=1;count<=num_chans;count++)
                            mess_c("----------");
                       out_clrf();
                       out_clrf();
                       count2=0;
                       }  /* if */
                  count2++;
                  pointer = adin;
                  ad_in_a(pointer,12,1);
                  pointer=adin+12;
                  ad_in_b(pointer,12,1);
                  for (count=1;count<=num_chans;count++) {
                       out_int(adin[ch_array[count]]);
                       mess_c("    ");
                       } /* for */
                  out_clrf();
                  if ((cntrl=='P') | (con_dis == 'D')) {
                       mess_c("--PAUSED--");
                       c=wait_char();
                       out_clrf();
                       }  /* if */
              } /* while */
      } /* if   */
/* main */  }
```

APPENDIX C

DARAMP

```
/*
** Routine to output a ramp to the D/A's
*/

extern  int     in_char(),out_crlf(),in_int(),mess_c(),da_out();
extern  int     cio_unit();
        int     start_chan,stop_chan,i,chan,num_chans;
        int     ch_array[20],j,data_array[1],*k;
        int     start_count,stop_count,delta_count;
        char    c,d;

main()
{
        cio_unit();
        out_crlf();
        mess_c("D/A ramp program ");
        out_crlf();

        mess_c("Input start count :");
        start_count=in_int();
        out_crlf();

        mess_c("Input stop count  :");
        stop_count=in_int();
        out_crlf();

        mess_c("Input delta count :");
        delta_count=in_int();
        out_crlf();

        /* get the channel definitions */

        mess_c("Output to how many channels ? (1-10 hex) ");
        num_chans=in_int();

        if (num_chans > 16)
           num_chans=16;

        if (num_chans < 1 )
           num_chans=1;
        out_crlf();

        for(j=0; j<=(num_chans-1); j++)
        {
           mess_c("Enter channel number :");
           ch_array[j]=in_int();
           out_crlf();
        }

        for(j=0; j<=15; j++)
        {
           data_array[0]=0;
           da_out(data_array,j,1);
```

```
         }

         /* main routine follows */

         mess_c("starting ramp ");
         out_crlf();

         while( ((c=in_char())!=0x1b) & (d!=0x1b) )
         {
            for(i=start_count; i<=stop_count; i+=delta_count)
            {
               for(j=0 ; j<=(num_chans-1) ; j++)
               {
                  da_out(data_array,ch_array[j],1);
                  data_array[0]=i;
                  d=in_char();
               }
            }
         }

} /* MAIN() */
```

APPENDIX D

ADTODA

```
/*
**  ad_to_da.c :
**               program to routine an A/D channel
**               to a D/A channel
*/

extern  int  da_out(),ad_in(),mess_c(),in_char(),out_crlf(),in_int();
extern  int  out_int(),out_char();
        int  da_chan,ad_chan,adin[30];
        char c,d;

main()
{
        out_crlf();
        mess_c("A/D to D/A routing program ");
        out_crlf();
        out_crlf();

        mess_c("A/D channel number :");
        ad_chan = in_int();
        out_crlf();

        mess_c("D/A channel number :");
        da_chan = in_int();
        out_crlf();

        if( da_chan>15 )
           da_chan = 15;

        mess_c("Routing started :");
        out_crlf();
        out_crlf();

        while( (c=in_char())!=0x1b )
        {
           ad_in(adin);
           da_out(da_chan,adin[ad_chan]);
        }

        out_crlf();
        out_crlf();

} /* MAIN() */
```

APPENDIX E

DSCOUT

```
/*
**  Program to test the discrete output
**
*/

extern  int     in_char(),out_int(),mess_c(),out_crlf();
extern  int     cio_unit(),cio_outa();
        int     b,i;
        char    c;

main()
{
        cio_unit();
        mess_c("Discrete output test routine ");
        out_crlf();
        out_crlf();
        mess_c("Starting sequence ");
        out_crlf();
        while( (c=in_char())!=0x1b)
        {
           b = 1;
           while( (b<=128) & ((c=in_char())!=0x1b) )
           {
              cio_outa(b);
              b *= 2;
              for(i=0; i<=32766; i++)
                 ; /* delay */
           }
           /* now blink and invert */
           b = 1;
           while( (b<=128) & ((cin=in_char())!=0x1b) )
           {
              i = b^0xff;
              cio_outa(i);
              b *= 2;
              for(i=0; i<=32766; i++)
                 ; /* delay */
           }
        }

} /* MAIN() */
```

APPENDIX F

ACTIVITY

```
/* routine to check out the activity circuitry */

extern  int  cio_init(),cio_outb(),cio_outc(),mess_c(),out_crlf();
        int  i,j,k,data[3],*point;
        char c,d,e;

main()
{
   cio_init();
   out_crlf();
   mess_c("Starting activity test...");
   out_crlf();
   for (j=0; j<=3; j++)
   {
      mess_c("D/A channel ");
      out_int(j);
      mess_c(" test");
      out_crlf();
      for (i=0; i<=16384; i++)
      {
         data[0]=0x4000;
         point= data;
         da_out(point,j,1); /* address=point, channel=j, data's=1 */
      }
   }

   out_crlf();
   mess_c("All channels ");
   out_crlf();

   while (1)
   {
      for (i=0; i <= 32766; i++)
      {
         c = in_char();
         j = 0;
         point = data;
         da_out(point,j,4);

         if( c==0x1b )
            break;
      }

      if( c==0x1b )
         break;
   }

   /* reset all the activity lights */

 } /* main */
```

DISTRIBUTION

	No. of Copies
University of Alabama in Huntsville Off Campus Research Facility ATTN: AMSMI-RD-SS-SE, Bldg 5400	10
US Army Materiel System Analysis Activity ATTN: AMXSY-MP Aberdeen Proving Ground, MD 21005	1
ITT Research Institute ATTN: GACIAC 10 W. 35th Street Chicago, IL 60616	1
AMSMI-RD, Dr. McCorkle	1
Dr. Rhoades	1
AMSMI-RD-CS-R	15
AMSMI-RD-CS-T	1
AMSMI-RD-SS-SE, Dr. Hallum	1
AMSMI-GC-IP, Mr. Bush	1

END

6-87

DTIC